R

Alejandra E. Medina Rivera
Licenciatura en Ciencias Genómicas.
Centro de Ciencias Genómicas, UNAM

Cuernavaca, Mexico
Feb, 2010

Introduction to R and Statistics

R beginnings

R is a dialect of the S language.
S is a language that was developed by John Chambers and others at Bell Labs in 1976. It was initiated as an internal statistical analysis environment.
R was created in 1991 in New Zeland by Ross Ihaka and Robert Gentleman.

- Syntax is very similar to S.
- As S is an interpretative language in an interpretative environment.
- Runs on almost any standard computing platform/OS (even PDAs and PlayStation 3)
- Graphic capabilities very sophisticated and better than most stat packages.
- It's an open source implementation of S. It's Free !!!!!.

Free Software

With the *free software*, you are granted

- The freedom to run the program, for any purpose
- The freedom to study how the program works, and adapt it to your needs.
- The freedom to redistribute copies so you can help your neighbor.
- The freedom to improve the program, and release you improvements to the public, so that the whole community benefits.

The R environment

R is an integrated suite of software for data manipulation, calculation and graphical display. Among other things it has:

- An effective data handling and storage facility.
- A suite of operators for calculations on arrays, in particular matrices.
- A large, coherent, integrated collection of intermediate tools for data analysis.
- Graphical facilities for data analysis and display.
- Simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
- R has a six month release cycle: stable and devel version.

R and statistics

Many people use R as a statistics system.
R is an environment within which many classical and modern statistical techniques have been implemented.
A few of these are built into the base R environment, but many are supplied as packages. There are about 25 packages supplied with R (called standard and recommended packages) and many more are available through the CRAN family of Internet sites (via `http://CRAN.R-project.org`) and elsewhere.

Install R

- For Windows and Mac, basically download the base binary from CRAN, double click on it and follow the instructions.
  - Windows stable and Mac stable releases.
- For Linux/Unix, it will depend on the flavor you have. Say you have Ubuntu, then you need to follow these instructions to get the latest stable version as `sudo apt-get install r-base` is generally not updated to the latest version.
- For this course you'll need the `R` version 2.12.0

Objects

Object-oriented programming is a style of programming that has become popular in recent years. Much of the popularity comes from the fact that it makes it easier to write and maintain complicated systems. Central to any object-oriented language are the concepts of *class* and of *methods*.

- A *class* is a definition of an object.
  - Typically a class contains several slots that are used to hold class-speci
    c information.
  - An object in the language must be an instance of some class.
  - Programming in `R` is based on objects or instances of classes.

Objects

- Computations are carried out via *methods*. *Methods* are basically functions that are specialized to carry out specific calculations on objects, usually of a specific class.
- Another feature of most object-oriented languages is the concept of *inheritance*. In most programming problems there are usually many objects that are related to one another. The programming is considerably simplied if some components can be reused.If a class inherits from another class then generally it gets all the slots in the parent class and can extend it by adding new slots.

Calling R

To access R there are several options:

- In Windows, doble click on the R icon.
- In Mac, doble click on the R icon.
- For any version of Linux/Unix, Mac and Windows type R on the console/terminal, this opens R directly.
- Using a R script.
  On a text editor (Word is NOT one) write the program and latter use it calling it with the `source()` function.
  You can insert comments into your code by using the # symbol.

Calling R

- Through Emacs and ESS.
  There are several version of Emacs depending on the platform Emacs detects with ESS automatically when a R script is in use recognizing the *.R* extension and calls the interpreter. This option is highly recommended. At the very least use a text editor and copy/paste your commands

One important command is the one that gets you out of the R environment in the console/terminal

- `q()` command to exit R.

R: An overgrown calculator

The simplest possible task on R is to enter an arithmetic expression and receive a result

- Addition

```
> 2 + 2
[1] 4
```

- Multiplication

```
> 2 * 3
[1] 6
```

- Division

```
> 2/3
[1] 0.6666667
```

- Exponents

R: An overgrown calculator

```
> 2^3
[1] 8
```

- Logarithm

```
> log(3, base = 2)
[1] 1.584963
```

- Square root

```
> sqrt(2)
[1] 1.414214
```

- Pi

```
> pi
[1] 3.141593
```

Workspace and history

All variables created in R are stored in a common workspace, you can see all this variables with the function ls(). You can erase from this workspace any variable using the command rm().

Sometimes you need to interrupt your work, so saving your R session, objects and/or history is useful.

- You can save and load objects by specifying the objects, path and file name into a .Rda file.

```
> save(object1, object2, file = file.path("folder
+      "file.Rda"))
> load(file = file.path("folder",
+      "file.Rda"))
```

Workspace and history

- To view your recent commands use the history function. You can save and load your history using savehistory and loadhistory.

```
> history()
> savehistory(file = file.path("folder",
+       "file.Rhistory"))
> loadhistory(file = file.path("folder",
+       "file.Rhistory"))
```

- You can save your session into a `.Rdata` file by specifying so when quitting or by using the save.image function and use load to reload it.

Workspace and history

```
> q(save = "yes")
> save.image(file = file.path("folder",
+      "file.Rdata"))
> load(file = file.path("folder",
+      "file.Rdata"))
```

- While working, you might need to change your working directory or view what's in there. Functions such as getwd, setwd, list.files() and dir() will be most helpful.

Searching for Help

- THE function for getting help is `help()`
  For example, lets say you don't know what the names function does, so you can get info about this using `help("names")` or some other short ways: `?names` and `?"names"`.
- For a deeper search you can use `help.search()`, this function looks inside the manuals for a word or words. Example, `help.search("names")`
- If you are looking for a function and you are not sure of the name use `apropos()`. Example, `apropos("names")`. Other usefull functions are:
  - `help.start()`,
  - `RSiteSearch()`,
  - `args()`

Searching for Help

  - `example()`.
- An other option is to be included on the help mail list, where you can ask more specific questions. They'll request information regarding your session. How can you obtain this information?

Basics on Packages

An R installation contains a library of packages. Some of these packages are part of the basic installation. Other can be downloaded from CRAN (via `http://CRAN.R-project.org`). You can even create your own packages.
A package can contain functions written in the R lenguage, dynamically loaded libraries of compiled code (written in `C` and `Fortran` mostly), and data sets. CRAN R packages can be downloaded using the function install.packages().

```
> install.packages("ISwR")
```

To load the content of the library we use library()

```
> library(ISwR)
```

Basics on Packages

The loaded packages are not considered part of the user workspace. If you terminate your R session and start a new session with the saved workspace, then you will have to load them again. To .^{er}ase.^{a} pacakge from the current session you can use the function detach(”package:namePackage”)

Data Types

R has a rich set of self-describing data structures.

```
> z <- "z"
> class(z)

[1] "character"
```

There is no need to declare the types of the variables.

Data Structures

The principal data structures in R are:

- `vector`- array of objects of the same type
- `matrix`- array of vectors
- `list`- can contain objects of different types
- `environment`- hashtable
- `data.frame`-array of vectors, lists or both.
- `factor`- categorical
- `fucntion`

Packages as `Bioconductor` provide other types of data structures.

Atomic Data Structures

In R, the basic data types are vectors, not scalars
A vector contains an index set of values that are all of the same type:

- *logical*
- *numeric*
- *complex*
- *character*

The numeric type can be further broken down into *integer*, *single*, and *double* types (but this is only important when making calls to foreign functions, eg. C or Fortran)

Value Assignation

- In R, there are two principal ways to assign values to variables: = and <-
- The most used one is <- to avoid confusion since the symbol = is used to assign values inside functions.

```
> A <- c(a = 1, b = 2)["b"]
> A = c(a = 1, b = 2)["b"]
> A

b
2
```

Variables - Vectors

- Remember R is a vector language, this means all variables are vectors.
- R es `c()` is a useful function. You can create a vector containing different types of variables.

```
> v1 <- c(1:10)
> v2 <- runif(10)
> v3 <- sample(c("A", "C", "G", "T"),
+      size = 10, replace = TRUE)
> v4 <- v3 %in% c("A", "G")
> v5 <- c("foo", 2, TRUE)
> v6 <- c(2, "3")
```

What's stored on v5?

Variables - Vectors

- mode function indicates the type of variable contained in the vector.
- as function can change the type of the vector.
  Example: change the *numeric* mode of vectors `v5` y `v6`:

```
> mode(v5)
> as.numeric(v5)
> as.numeric(v6)
> help(as.vector)
> help(as)
```

Vectorized arithmetic

You can do calculations with vectors just like ordinary numbers, as long as they are of the same length. Lest assume we know the weight and hight of a group of persons, and try to obtain their BMI.

```
> weight <- c(50, 60, 57, 72, 90,
+     95)
> height <- c(1.6, 1.7, 1.65, 1.72,
+     1.9, 1.85)
> bmi <- weight/height^2
> bmi

[1] 19.53125 20.76125 20.93664 24.33748
[5] 24.93075 27.75749
```

Vectorized arithmetic

Note that the operations are carried out element-wise, that is, the first value of bmi is

$$65/1{,}75^2 \tag{1}$$

Vectors Recycling

- In `R` most fo the functions are vectorize. Example: `x = 2; y = 3; x + y` is in fact $x[i] + y[i], i \in 1, ...\mathtt{max}\{|x|, |y|\}$
- If the length of to vectors is not the same, `R` recycles the shortest till it reaches the length of the longest.

Example (Recycling)

```
c(2,3) + c(3,4,5)
> c(2, 3) + c(3, 4, 5)
[1] 5 7 7
```

and compare it with `c(2,3) + c(3,4,5,8)`

```
> c(2, 3) + c(3, 4, 5, 8)
[1]  5  7  7 11
```

Functions that create vectors

We have already used some of them, but just to be sure lest mention them all

- c(), this concatenates values into a vector
- seq(), creates a sequence of numbers
- rep(), repeat numbers
- runif and al the rSomething, create vectors selecting random numbers from a distribution.

```
> c(42, 57, 12, 39)

[1] 42 57 12 39

> seq(4, 9)

[1] 4 5 6 7 8 9

> seq(4, 10, 2)
```

Functions that create vectors

```
[1]  4  6  8 10
```

```
> seq(0, 1, by = 0.1)
```

```
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8
[10] 0.9 1.0
```

What exactly does the `seq()` function do?

```
> oops <- c(7, 9, 13)
> rep(oops, 3)
```

```
[1]  7  9 13  7  9 13  7  9 13
```

```
> rep(1:2, c(10, 15))
```

```
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2
[19] 2 2 2 2 2 2 2
```

Functions that create vectors

```
> rep("Small", 3)

[1] "Small" "Small" "Small"

> c(rep("Small", 3), rep("Medium",
+     4))

[1] "Small"  "Small"  "Small"  "Medium"
[5] "Medium" "Medium" "Medium"
```

Matrices and arrays

- A matrix is a two dimensions array .
- Matrix notation is extended to all other data structures in R.
- Matrices are vectors with dimension.

```
> x <- 1:12
> dim(x) <- c(3, 4)
> x

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Matrices and arrays

Create Matrices

In R there are several way to create matrices, the principal one is using the function `matrix()`

```
> matrix(1:12, nrow = 3, byrow = T)

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12

> x <- matrix(1:12, nrow = 3, byrow = T)
> rownames(x) <- LETTERS[1:3]
> x
```

Matrices and arrays

```
  [,1] [,2] [,3] [,4]
A    1    2    3    4
B    5    6    7    8
C    9   10   11   12
```

Which is the result f the following commands?

```
> t(x)
> m2 <- matrix(c(1, 3, 2, 5, -1,
+      2, 2, 3, 9), ncol = 3, byrow = T)
> m2[-1, -1]
```

Create Matrices

An other way to create a matrix is attaching vectors, using them as columns or lines of the matrix.
Fucntions for this are: `rbind()` and `cbind()`

Matrices and arrays

```
> cbind(A = (1:4), B = (5:8), C = (9:12))

     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12

> rbind(A = (1:4), B = (5:8), C = (9:12))

  [,1] [,2] [,3] [,4]
A    1    2    3    4
B    5    6    7    8
C    9   10   11   12
```

Matrices

- A vector has no dimensions, but if you assign dimensions to a vector you can turn it into a matrix [1]. This affects how the content is stored inside the object:

```
> V <- runif(100)
> print(V[1:9])

[1] 0.4662715 0.6578328 0.3689447
[4] 0.8913764 0.4859088 0.1035492
[7] 0.8401884 0.5745480 0.8115455

> dim(V)

NULL

> dim(V) <- c(2, 5, 10)
> print(V[, , 1])
```

Matrices

```
          [,1]      [,2]      [,3]
[1,] 0.4662715 0.3689447 0.4859088
[2,] 0.6578328 0.8913764 0.1035492
          [,4]      [,5]
[1,] 0.8401884 0.8115455
[2,] 0.5745480 0.5540904
```

[1] array of two or more dimensions

Operations with Matrices

Matrix computation is usually done component-wise.

```
> m1 <- matrix(1:4, ncol = 2)
> m2 <- matrix(c(10, 20, 30, 40),
+      ncol = 2)
```

- Scalar multiplication

```
> 2 * m1
```

```
     [,1] [,2]
[1,]    2    6
[2,]    4    8
```

- Matrix addition

```
> m1 + m2
```

Operations with Matrices

```
     [,1] [,2]
[1,]   11   33
[2,]   22   44
```

- Component-wise multiplication

  > *m1 * m2*

```
     [,1] [,2]
[1,]   10   90
[2,]   40  160
```

- Matrix multiplication

  > *m1 %*% m2*

```
     [,1] [,2]
[1,]   70  150
[2,]  100  220
```

Operations with Matrices

- Inverse Matrix

  ```
  > solve(m1)
  ```

  ```
       [,1] [,2]
  [1,]   -2  1.5
  [2,]    1 -0.5
  ```

  You can find all necessary functions to do all the operations you can need on a matrix like the eigenvalues,. Can you tell me which is the function to get the eigenvalues?

Categories=Factors

- Data in statistics is usually classified.
- The variables that let us store categorized data are called *factores*.

```
> pain <- c(0, 3, 2, 2, 1)
> fpain <- factor(pain, levels = 0:3)
> levels(fpain) <- c("none", "mild",
+      "medium", "severe")
> fpain
```

```
[1] none   severe medium medium mild
Levels: none mild medium severe
```

What will happen with the factor if I use `as.numeric()` on it?

```
> levels(fpain)
```

Categories=Factors

```
[1] "none"   "mild"   "medium" "severe"
> as.numeric(fpain)
[1] 1 4 3 3 2
```

If you don't specify the `levels` in the `factor()` functions, the levels will be taken from the sorted unique values represented in the vector. Don't forget this cause you'll need this tip on the future.

Lists

- Data in statistics tends to be classified or subdivided.
- Lists are an easy way to combine different objects in one.
- Remember we can store categorized date in *factores*.

Numeric Reference

Elements on a list are always *enumerated*. If `Lst` is a list with four elements, one element is `Lst[[4]]` and if this element is a vector you can access the first element using: `Lst[[4]][1]`

Name Reference

Elements on a list can be access by name: `list$name`

Lists

```
> Lst <- list(name = "Fred", wife = "Mary",
+      no.children = 3, child.ages = c(4,
+          7, 9))
> Lst$name

[1] "Fred"

> Lst[[1]]

[1] "Fred"

> Lst$wife

[1] "Mary"

> Lst$child.ages[1]

[1] 4
```

Lists

```
> Lst[[4]][1]
[1] 4
> length(Lst)
[1] 4
```

Data Frame

Data Frames

In R *data frames* are very important objects. A `data.frame` is a table composed by one or more vectors and/or factors of the same length and different data types.

- `dataframe$variable` o `dataframe[["variable"]]`.
- Functions `attach` and `detach` can add variables from a `data frame` to the R environment.[2]; function `with(data.frame, command)` does the same.
- You can display the first and last elements of a `data.frame` or *array* using functions `head()` or `tail()`.

[2]Not recommendable if you are going to modify values of the `data frame` or if you have variables with the same names

Read Files

The principal function in R to read files is `read.table()`.

Read a `data.frame` from a file

The `read.table()` function reads a table from a file and stores it into a `data.frame` if :

- The first line is a header with the name of the variable in each column of the `data.frame`. If the header is not provided R automatically assigns variables `V1,V2...Vn` to each column.
- Each line has to have a unique ID, `row.name`
- If a `data.frame` is not necesary, you can change the format using function `as.format()` .

Read Files

Example

Read table

```
> arch <- "/Users/amedina/Documents/CCG/Cursos/Compu_
> heartatk <- read.table(file = arch,
+       header = TRUE)
```

- Other useful functions to read files are:
  - `scan()`
  - `read.table()`,
  - `read.csv()`
  - `source()`.
- to learn more: `help(read.csv)`
- `scan()` is useful when you don't know the structure of your data..

Read Files

- `source()` command used to read scripts `R` and execute them inside the current session.

Directories

- Sometimes you want to access several files from the same directory of folder
- Maybe you don't want to open all of them but some, so you have to look for a common pattern in the names.
- The automatic way to do it:

```
> setwd("/Users/amedina/Documents/CCG/Cursos/Compu_St
> files <- list.files(pattern = ".txt")
> for (i in files) {
+     x <- read.table(i, header = TRUE)
+ }
```